Jon

Network Working Group
Request for Comments #48

J. Postel
S. Crocker
21 April 70
UCLA

A Possible Protocol Plateau

I Introduction

We have been engaged in two activities since the network meeting of March 17, 1970 and, as promised, are reporting our results.

First, we have considered the various modifications suggested from all quarters and have formed preferences about each of these. In section II we give our preferences on each issue, together with our reasoning.

Second, we have tried to formalize the protocol and algorithms for the NCP, we attempted to do this with very little specification of a particular implementation. Our attempts to date have been seriously incomplete but have led to a better understanding. We include here, only a brief sketch of the structure of the NCP. Section III gives our assumptions about the environment of the NCP and in Section IV the components of the NCP are described.

II Issues and Preferences

In this section we try to present each of the several questions which have been raised in recent NWG/RFC's and in private conversations, and for each issue, we suggest an answer or policy. In many cases, good ideas are rejected because in our estimation they should be incorporated at a different level.

A. Double Padding

As BBN report #1822 explains, the Imp side of the Host-to-Imp interface concatenates a 1 followed by zero or more 0's to fill out a message to an Imp word boundary and yet preserve the message length. Furthermore, the Host side of the Imp-to-Host interface extends a message with 0's to fill out the message to a Host word boundary.

BBN's mechanism works fine if the sending Host wants to send an integral number of words, or if the sending Host's hardware is capable of sending partial words. However, in the event that the sending Host wants to send an irregular length message and its hardware is only capable of sending word-multiple messages, some additional convention is needed.

One of the simplest solutions is to modify the Imp side of the Host-to-Imp interface so that it appends only 0's. This would mean that the Host software would have to supply the trailing 1. BBN rejected the change because of an understandably strong bias against hardware changes. It was also suggested that a

five instruction patch to the Imp program would remove the interface supplied 1, but this was also rejected on the new grounds that it seemed more secure to depend only upon the Host hardware to signal message end, and not to depend upon the Host software at all.

Two other solutions are also available. One is to have "double padding", whereby the sending Host supplies 10* and the network also supplies 10*. Upon input, a receiving Host then strips the trailing 10* 10*. The other solution is to make use of the marking. Marking is a string of the form 0*1 inserted between the leader and the text of a message. The original intent of marking was to extend the leader so that the sending Host could begin its text on a word boundary. It is also possible to use the marking to expand a message so that it ends on a word boundary.

Notice that double padding could replace marking altogether by abutting the text beginning against the leader. For 32 bit machines, this is convenient and marking is not, while for other lengths, particularly 36 bit machines, marking is much more convenient than double padding.

We have no strong preference, partially because we can send word fragments. Shoshani, et al in NWG/RFC #44 claim that adjusting the marking does not cause them any problems, and they have a 32 bit machine. Since the idea of marking has been accepted for some time, we suggest that double padding not be used and that marking be used to adjust the length of a message. We note that if BBN ever does remove the 1 from the hardware padding, only minimal change to Host software is needed on the send side.

A much prettier (and more expensive) arrangement was suggested by W. Sutherland. He suggested that the Host/Imp interfaces be smart enough to strip padding or marking and might even parse the message upon input.

B. Reconnection

A very large population of networkers has beat upon us for including dynamic reconnection in the protocol. We felt it might be of interest to relate how it came to be included.

After considering connections and their uses for a while, we wondered how the mechanism of connections compared to existing forms of intra-Host interprocess communication. Two aspects are of interest, what formalisms have been presented in the literature, and what mechanisms are in use. The formalisms are interesting because they lead to uniform implementation and parsimonious design. The existing mechanisms are interesting because they point out which

problems need solving and sometimes indicate what an appropriate formalism might be. In particular, we have noticed that the mechanisms for connecting a console to the logger upon dial in, the mechanisms for creating a job, and the mechanisms for passing a console around to various processes within a job tend to be highly idiosyncratic and distinct from all other structures and mechanisms within an operating system.

With respect to the literature, it appears there is only one idea with several variations, viz processes should share a portion of their address spaces and cooperatively wake up each other. Semaphores and event channels are handy extensions of wake up signals, but the intent is basically the same. (Event channels could probably function as connections, but it seems not to be within their intended use. In small systems, the efficiency and capacity of event channels are inversely related.)

With respect to existing implementations, we note that several systems allow a process to appear to be a file to another process. Some systems, e.g. the SDS-940 at SRI impose a master/slave relationship between two processes so connected, but other systems provide for a coequal relationship e.g. the AI group's PDP-6 system at MAC. The PDP-6 system also has a feature whereby a superior process can "surround" an inferior process with a mapping from device and file names to other device and file names. Consoles have nearly the same semantics as files, so it is quite reasonable for an inferior process to believe it is communicating with the console but in fact be communicating with another process.

The similarity between network connections and existing sequential interprocess connections supports our belief that network connections are probably the correct structure for using the network. Moreover, the structure is clean enough and compatible with enough machines to pass as a formalism or theory, at least to the extent of the other forms of interprocess communication presented in the literature.

Any new formalism, we believe, must meet at least the following two tests.

1. What outstanding problems does it solve?
2. Is it closed under all operations?

In the case of network connections, the candidates for the first are the ones given above viz all operations involving connecting a console to a job and or a process. Also of interest are the

modelling of sequential devices such as tape drives, printers and card readers, and the modelling of their buffering (spooling, symbiont) systems.

The second question mentions closure. In applying the connection formalism to the dial-in and login procedures, we felt the need to include some sort of switching or reconnection, and an extremely mild form is presented in an SJCC paper, which is also NWG/RFC #33. This mild form permits only the substitution of AEN's, and even then only at the time of connection establishment. However, it is a common experience that if an operation has a natural definition on an extended domain, it eventually becomes necessary or at least desirable to extend its definition. Therefore, we considered the following extensions.

1. Switching to any other socket, possibly in another Host.
2. Switching even after data flow has started.

There is even some precedent for feeling these extensions might be useful. In one view of an operating system, we see all available phone lines as belonging to a live process known as the logger. The logger answers calls, screens users, and creates jobs and [illegible] processes. One of the features of most telephone answering equipment is that many phone lines may serve the same phone number by using a block of sequential numbers and a rotary answering system. In our quest for accurate models of practical systems, we wanted to be able to provide equivalent service to network users, i.e. they should be able to call a single advertised number and get connected to the logger. Thus a prima facie case for switching is established.

Next we see that after the logger interrogates a prospective user, it must connect the user to a newly created job. Data flow between the user and the logger has already commenced, so flow control has to be meshed with switching if it is desired not to lose or garble data in transit.

With respect to inter-Host switching, we find it easy to imagine a utility service which is distributed throughout the network and which passes connections from one socket to another without the knowledge of the user. Also, it is similar to the more sophisticated telephone systems, to standard facilities of telephone company operators, and to distributed private systems.

These considerations led us to investigate the possibility of finding one type of reconnection which provided a basis for all known models. The algorithm did not come easily, probably because of inexperience with finite state automata theory, but eventually we produced the algorithm presented in NWG/RFC #36. A short time later, Bill Crowther produced an equivalent algorithm which takes an alternate approach to race conditions.

Networkers seem to have one of two reactions. Either it was pretty and (perhaps ipso facto) useful, or it was complex and (again perhaps ipso facto) unnecessary. The latter group was far more evident to us, and we were put into the defensive position of admitting that dynamic reconnection was only

1. pretty
2. useful for login and console passing

In response to persistant criticism, we have made the following change in the protocol. Instead of calling socket <O,H,O> to login, sockets of the form <U,H,O> and <U,H,1> are the input and output sockets respectively of a copy of the logger or, if a job has been started with user id U, these sockets are the console sockets. The protocol for login is thus to initiate a connection to <U,H,O> and <U,H,1>. If user id U is not in use, a copy of the logger will respond and interrogate the caller. If user id U is in use, the call will be refused. This modification was suggested by Barry Wessler recently. (Others also suggested this change much earlier; but we rejected it then.)

The logger may demand that the caller be from the same virtual net, i.e. the caller have user id U in some other Host, or it may demand that the user supply a password matched to user id U, or it may demand both. Some systems may even choose to permit anybody to login to any user id.

After login, AEN's 0 and 1 remain the console AEN's. Each system presumably has mechanisms for passing the console, and these would be extended to know about AEN's 0 and 1 for network users. Passing the console is thus a matter of reconnecting sockets to ports, and happens within the Host and without the network.

In conversation with Meyer and Skinner after NWG/RFC #46 was received, they suggested a login scheme different from both Meyer's and ours in section above. Their new scheme seemed a little better and we look forward to their next note.

It is generally agreed that login should be "third-level", that is, above the NCP level. We are beginning to be indifferent about particular logins schemes; all seem ok and none impress us greatly. We suggest that several be tried. It is some burden, of course, to modify the local login procedure, but we believe it imposes no extra hardship to deal with diverse login procedures. This is because the text sequences and interrupt conventions are so heterogenous that the additional burden of following, say, our scheme on our system and Meyer's on Multics is minimal.

We are agreed that reconnection should not be required in the initial protocol, and we will offer it later as an optional and experimental tool. In addition, we would like to be on record as predicting that general reconnection facilities will become useful and will provide a unifying framework for currently ad hoc operating system structures.

C. Decoupling Connections and Links

Bill Crowther (BBN) and Steve Wolfe (UCLA) independently have suggested that links not be assigned to particular connections. Instead, they suggest, include the destination socket as part of the text of the message and then send messages over any unblocked link.

We discussed this question a little in NWG/RFC #37, and feel there is yet an argument for either case. With the current emphasis on simplicity, speed and small core requirements, it seems more efficient to leave links and connections coupled. We, therefore, recommend this.

D. Error Reporting

As mentioned by J Heafner and E. Harslem of RAND, it is important to treat errors which might occur. A good philosophy is to guard against any input which destroys the consistency of the NCP's data base.

The specific formulation of the error command given by Heafner and Harslem in NWG/RFC #40 and by Meyer in NWG/RFC #46 seems reasonable and we recommend its adoption. Some comments are in order, however.

A distinction should be made between resource errors and other types of errors. Resource errors are just the detection of overload conditions. Overload conditions are well-defined and valid, although perhaps undersirable. Other types of errors reflect errant software or hardware. We feel that resource errors should not be handled with error mechanisms, but with mechanisms specific to the problem. Thus the <CLS> command may be issued when there is no more room to save waiting <RFC>'s. Flow control protocol is designed solely to handle buffering overload.

With respect to true errors, we are not certain what the value of the <ERR> command is to the recipient. Presumably his NCP is broken, and it may only aggravate the problem to bombard it with error commands. We therefore, recommend that error generation be optional, that all errors be logged locally in a chronological file and that <ERR> commands received likewise be logged in a chronological file. No corrective action is specified at this time.

In the short time the network has been up at UCLA, we have become convinced that the network itself will generate very few errors. We have watched the BBN staff debug and test the IMP program, and it seemed that most of the errors affected timing and throughput rather than validity. Hence most errors will probably arise from broken Hosts and/or buggy NCP's.

E. Status Testing and Reporting

A valuable debugging aid is to be able to get information about what a foreign NCP thinks is happening. A convenient way to do this is to permit NCP's to send status whenever they wish, but to always have them do it whenever they receive a request.

Since we view this feature as primarily a debugging tool, we suggest that a distinct link, link 255, be used. The intent is that the processing of status requests and generating of status messages should use as little of the normal machinery as possible. Thus we suggest that link 255 be used to send "request status" and "status is" commands. The form follows the suggestion on page 2 of NWG/RFC #40.

Meyer's <ECO> command is easily implemented and serves the more basic function of testing whether a foreign NCP is alive. We suggest that the length of the <ECO> command be variable, as there seems to be no significance in this context to 48 bits. Also, the value of a (presumably) 8 bit binary switch is unclear, so we recommand a pair of commands:

```
                <ECO>   <length>    <text>
and
                <ERP>   <length>    <text>
where
                <length> is 8 bits.
```

Upon receipt of an <ECO> command the NCP would echo with the <ERP> command.

J. Postel
S. Crocker
21 April 70

F. Expansion and Experimentation

As Meyer correctly points out in NWG/RFC #46, network protocol is a layered affair. Three levels are apparent so far.

1. IMP Network Protocol
2. Network Control Program Protocol
3. Special User level or Subsystem Level Protocol

This last level should remain idiosyncratic to each Host (or even each user). The first level is well-specified by BBN, and our focus here is on level 2. We would like to keep level 2 as neutral and simple as possible, and in particular we agree that login protocol should be as much on level 3 as possible.

Simplicity and foresight notwithstanding, there will arise occasions when the level 2 protocol should change or be experimented with. In order to provide for experimentation and change, we recommend that only link numbers 2 through 31 be assigned to regular connections, with the remaining link numbers, 32 to 255, used experimentally. We have already suggested that link 255 be used for status requests and replies, and this is in consonance with our view of the experimental aspects of that feature.

We also recommend that control command prefixes from 255 downward be used for experimentation.

These two conventions are sufficient, we feel to permit convenient experimentation with new protocol among anv subset of the sites. We thus do not favor inclusion of Ancona's suggestion in NWG/RFC #42 for a message data type code as the first eight bits of the text of a message.

G. Multiplexing Ports to Sockets

Wolfe in NWG/RFC #38 and Shoshani et al in NWG/RFC #44 suggest that it should be possible to attach more than one port to a socket. While all of our diagrams and prototypical system calls have shown a one-one correspondence between sockets and ports, it is strictly a matter of local implementation. We note that sockets form a network-wide name space whose sole purpose is to interface between the idiosyncratic structures peculiar to each operating system. Our references to ports are intended to be suggestive only, and should be ignored if no internal structures corresponds them. Most systems do have such structures, however, so we shall continue to use them for illustration.

Network Working Group
Request for Comments #48

J. Postel
S. Crocker
21 April 70

H. Echoing, Interrupts and Code Conversion

1. Interrupts

We had been under the impression that all operating systems scanned for a reserved character from the keyboard to interpret it as an interrupt signal. Tom Skinner and Ed Meyer of MIT inform us that model 37 TTY's and IBM 2741 generate a "long space" of 200-500 milliseconds which is detected by the I/O channel hardware and passed to the operating system as an interrupt. The "long space" is not a character--it has no ASCII code and cannot be program generated.

Well over a year ago, we considered the problem of simulating console interrupts and rejected the <INT> type command because it didn't correctly model any system we knew. We now reverse our position and recommend the implementation of an INTERRUPT system call and an <INT> control command as suggested by Meyer in NWG/RFC #46.

Two restrictions of the interrupt facility should be observed. First, when communicating with systems which scan for interrupt characters, this feature should not be used. Second, non-console-like connections probably should not have interrupts. We recommend that systems follow their own conventions, and if an <INT> arrives for a connection on which it shouldn't the <INT> should be discarded and optionally returned as an error.

2. Echoing and Code Conversion

We believe that each site should continue its current echoing policy and that code conversion should be done by the using process. Standardization in this area should await further development.

Ancona's suggestion of a table-driven front-end transducer seems like the right thing, but we believe that such techniques are part of a larger discussion involving higher-level languages for the network.

I. Broadcast Facility

Heafner and Harslem suggest in NWG/RFC #39 a broadcast facility, i.e. <TER> and <BDC> . We do not fully understand the value of this facility and are thus disposed against it. We suspect that we would understand its value better if we had more experience with OS/360. It is probably true in general that sites running OS/360 or similar systems will find less relevance in our suggestions for network protocol than sites running time-shared systems. We would appreciate any cogent statement on the relationship between OS/360 and the concepts and assumptions underlying the network protocol.

J. Instance Numbers

Meyer in NWG/RFC #46 suggests extending a socket to include an instance code which identifies the process attached to the socket. We carefully arranged matters so that processes would be indistinguishable. We did this with the belief that both as a formal and as a practical matter it is of concern only within a Host whether a computation is performed by one or many processes. Thus we believe that all processes within a job should cooperate in allocating AEN's. If an operating system has facilities for passing a console from process to process within a job, these facilities mesh nicely with the current network protocol, even within reconnection protocol; but instance numbers interfere with such a procedure.

We suggest this matter be discussed fully because it relates to the basic philosophy of sockets and connections. Presently we recommend 40 bit socket numbers without instance codes.

K. AEN's

Nobody, including us, is particularly happy with our name AEN for the low order 8 bits of the socket. We rejected socket number, and are similarly unhappy with Meyer's socket code. The word socket should not be used as part of the field name, and we solicit suggestions.

III Environment

We assume that the typical host will have a time-sharing operating system in which the cpu is shared by processes.

Processes

We envision that each process is tagged with a user number. There may be more than one process with the same user number, and if so, they should all be cooperating with respect to using the network.

We envision that each process contains a set of ports which are unique to the process. These ports are used for input to or output from the process, from or to files, devices or other processes.

We also envision that each process has an event channel over which it can receive very short messages (several bits). We will use this mechanism to notify a process that some action external to the process has occurred.

To engage in network activity, a process attaches a local socket to one **one** of its ports. Sockets are identified by user number, host and AEN, and a socket is local to a process if their user numbers match and they are in the same host. A process need only specify an AEN when it is referring to a local socket.

Each port has a status which is modified by system calls and by concurrent events outside the process. Whenever the status of a port is changed, the process is sent an event over its event channel which specifies which port's status has changed. The process may then look at the port's status.

These assumptions are used descriptive material which follows. However, these assumptions are not imposed by the network protocol and the implementation suggested by section IV is in no way binding. We wish to make very clear that this material is offered only to provide clues as to what the implementation difficulties might be and not to impose any particular discipline.

For example, we treat <RFC>'s which arrive for unattached local sockets as valid, and queue them. If desired, an NCP may reject them, as Meyer suggests, or it might hold them for awhile and reject them if they're not soon satisfied. The offered protocol supports all these options.

Another local option is the one mentioned before of attaching multiple ports to a socket. We have shown one-one correspondence but this may be ignored. Similarly, the system calls are merely suggestive.

System Calls

These are typical system calls which a user process might execute. We show these only for completness; each site will undoubtably implement whatever equivalent set is convenient.

We use the notation

Syscall (arg$_1$, arg$_2$...; val$_1$...)

where

Syscall is the system call
arg$_1$ etc. are the parameters supplied with the call, and
val$_1$ etc. are any values returned by the system call.

Init (P,AEN,FS,Bsiz;C)

P	Specifies a port of the process.
AEN	Specifies a local socket. The user number of this process and host number at this host are implicit.
FS	Specifies a socket with any user number in any host, with any AEN.
Bsiz	Specifies the amount of storage in bits the user wants to devote to buffering messages.
C	The condition code returned.

Init attempts to attach the local socket specified by AEN to the port P and to initiate a connection with the socket FS. Possible returned values of C are

C = ok	The Init was legal and the socket FS is being contacted. When the connection is established or when FS refuses,the process ..will receive an event.
C = busy	The local socket was in use by a port on this or some other process with the same user number. No action was taken.
C = homosex	The AEN and FS were either both send or both receive sockets.
C = nohost	The host designation within FS isn't known.
C = bufbig	Bsiz is too large

Listen (P,AEN,Bsiz;C)

P	Specifies a port of the process.
AEN	Specifies a local socket.
Bsiz	Specifies a buffer size.
C	The returned legality code.

Codes for C are

C = ok
C = busy
C = bufbig

The local socket specifies by AEN is attached to P. If there is a waiting call, it is processed; otherwise no action is taken. When a call comes in, a connection will be established and the process notified via an event.

Close (P)

P Specifies a port of the process.

Any activity is stopped, and the port becomes free for other use.

Transmit (P,M,L1;L2,C)

P	Specifies port with an open connection.
M	The text to be transmitted.
L1	Specifies the length of the text.
L2	The length actually transmitted.
C	The error code.

Transmission between the processes on either side of the port takes place.

Codes for C are

C = ok

or

C = not open if no connection is currently open and otherwise uninhibited.

Status (P;C)

The status of port P is returned as C.

IV The NCP

We view the NCP as having five component programs, three associative tables, some queues and buffers, and a link assignment table. Each site will of course, vary this design to meet its needs, so our design is only illustrative.

The Component Programs

1. The Input Handler

   This is an interrupt driven input routine. It initiates Imp-to-Host transmission into a resident buffer and wakes up the Input Interpreter when transmission is complete.

2. The Output Handler

   This is as interrupt driven output routine. It initiates Host-to-Imp transmission out of a resident buffer and wakes up the Output Scheduler when transmission is complete.

3. The Input Interpreter

   This program decides whether the input is a regular message intended for a user, a control message, an Imp-to-Host message, or an error. For each class of message. This program takes the appropriate action.

4. The Output Scheduler

   Three classes of message are sent to the Imp

   (a) Host-to-Imp messages
   (b) Control messages
   (c) Regular messages

   We believe that a priority should be imposed among these classes. The priority we suggest is the ordering above. The Output Scheduler selects the highest priority message and gives it to the Output Handler.

5. The System Call Interpreter

   This program interprets requests from the user.

The two interesting components are the Input Interpreter and the System Call Interpreter. These are similar in that the Input Interpreter services foreign requests and the System Call Interpreter services local requests.

Associative Tables

We envision that the bulk of the NCP's data base is in three associative tables. By "associative", we mean that there is some lookup routine which is presented with a key and either returns successfully with a pointer to the corresponding entry, or fails if no entry corresponds to the key.

1. The Rendezvous Table

   "Requests-for-connection" and other attributes of a connection are held in this table. This table is accessed by local socket, but other tables have pointers to existing entries.

The components of an entry are:

(a) local socket (key)
(b) foreign socket
(c) link
(d) queue of callers
(e) text queue
(f) connection state
(g) flow state
(h) pointer to attached port

An entry is created when a user executes either an Init or a Listen system call or when a <RFC> is received. Some fields are unused until the connection is established, e.g. the foreign socket is not known until a <RFC> arrives if the user did a Listen.

2. The Input Link Table

The Input Interpreter uses the foreign host and link as a key to get a pointer to the entry in the rendezvous table for the connection using the incoming link.

3. The Output Link Table

In order to interpret RFNM's, the Input Interpreter needs a table in the same form as the Input Link Table but using outgoing links.

Link Assignment Table

This is a very simple structure which keeps track of which links are in use for each host. One word per host probably suffices.

Network Working Group
Request for Comments #48

J. Postel
S. Crocker
21 April 70

The following diagram is our conception of the Network Control Program. Boxes represent tables and Buffers, cylinders represent Queues and squishy ameoba like things represent component programs, the arrows represent data paths.

The abbreviated names have the following meanings.

ILT - Input Link Table

OLT - Output Link Table

LAT - Link Assignment Table

RT - Rendezvous Table

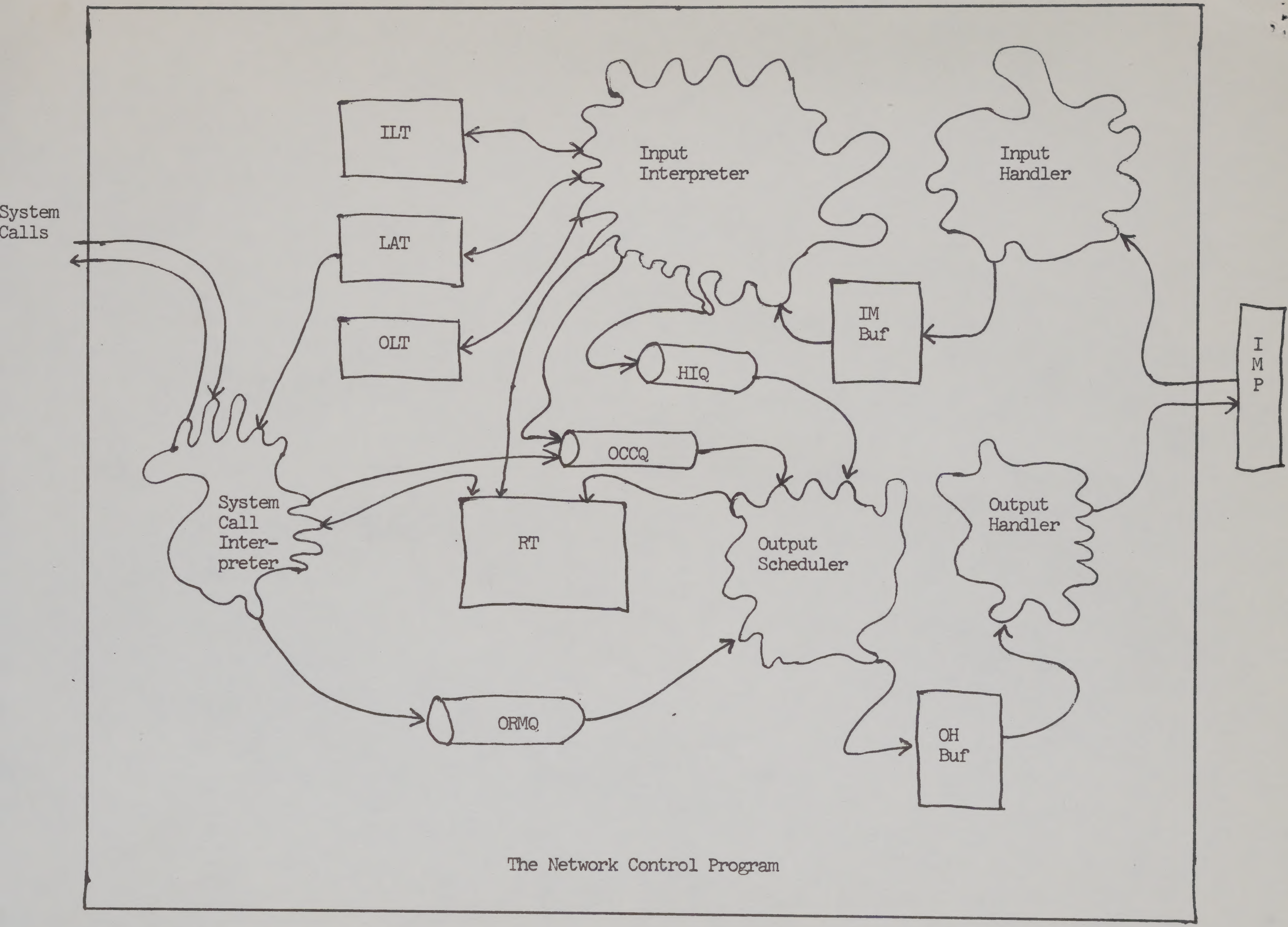
HIQ - Host to Imp Queue

OCCQ - Output Control Command Queue

ORMQ - Output Regular Message Queue

IHBuf - Buffer filled by the Input Handler from the IMP and emptied by the Input Interpreter

OHBuf - Buffer of outgoing messages filled from the Queues by the Output Scheduler and emptied by the Output Handler.



ILT
LAT
OLT
Input Interpreter
Input Handler
IM Buf
IMP
HIQ
OCCQ
RT
System Calls
System Call Inter-preter
Output Scheduler
Output Handler
ORMQ
OH Buf

The Network Control Program